



C quoi le C ?

Contents

1	Les bases	2
1.1	Jeux de main...	2
1.1.1	Kézako ?	2
1.1.2	Commençons par le début	3
1.1.3	Un fichier C tout bête	3
1.2	Les fonctions	3
1.3	Compilation	5
1.4	Les types de données	5
1.5	Afficher les résultats	6
1.6	Interactions	7
1.7	Travail manuel !	7
2	Tu peux pas test !	8
2.1	Découverte des tests	8
2.1.1	Les tests : qu'est-ce ?	8
2.1.2	if, else	8
2.1.3	Les opérateurs	9
2.1.4	Switch	10
2.2	Cherchons la boucle... d'or ?	11
2.2.1	while	11
2.2.2	for	12
3	Tableaux à la Picasso	13
4	Pointe pas du doigt !	14
4.1	Introduction aux pointeurs	14
4.1.1	Que c'est moche !	14
4.1.2	Mais comment ça marche alors ?	14
4.2	Pour un malloc libre !	15
4.2.1	malloc	15
4.2.2	free	15
4.2.3	Démonstration de l'utilité du free	16
4.3	Exemple simple sur les pointeurs	17
4.4	Les listes	18
5	Une main mieux employée	20
5.1	Pourquoi ? Mais pourquoiiiii ?	20
5.2	Retour sur la structure du main	20
5.3	Utilisation du main	21

Chapter 1

Les bases

1.1 Jeux de main...

1.1.1 Kézako ?

Tout programme codé en C contient une fonction **main**. Cette fonction est celle qui sera exécutée lorsqu'on lance le programme. Les autres fonctions écrites dans le code de l'application sont appelées via la fonction main.

Cette fonction a toujours la même structure. Afin de bien débiter, il est ingénieux de commencer chaque code C par les lignes suivantes :

```
1 int main(void) {  
2     return 0;  
3 }
```

et de compléter par ce qu'il faut. Ainsi, pas d'erreur de return oublié ou autres accolades bêtement mises deux fois (et ça arrive même aux meilleurs).



Tu trouves peut être, petit malin, que le fait de renvoyer l'int 0 à la fin de la fonction est totalement inutile. Tout d'abord, sache que c'est une convention fort utile par la suite.



Lors du TOP, on te demandera de renvoyer 0 si tout s'est bien passé lors de l'exécution de la fonction, et 1 si il y a eu une erreur (par exemple si l'abruti d'utilisateur a demandé à l'application d'ouvrir un fichier qui n'existe pas).

1.1.2 Commençons par le début

Au début de tout fichier C, on inclut les bibliothèques (ou librairies) contenant les fonctions nécessaires au reste du code. En fait, on va demander au compilateur de lire en premier lieu la notice, avant de se lancer dans le jeu. Cette notice s'appelle `stdio.h` et `stdlib.h`. Ces deux fichiers vont contenir des éléments essentiels au bon déroulement du programme (ainsi qu'à la compilation). Voici les deux premières lignes de tous tes codes C :

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

Chacune de ces bibliothèques va contenir des fonctions (par exemple **printf** que nous verrons plus loin) que vous pouvez utiliser directement.

1.1.3 Un fichier C tout bête

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     printf("hello world");
6     return 0;
7 }
```

Ce code contient bien les include de bibliothèques nécessaires, ainsi que la fonction `main`. Cette fonction retourne un `int` : 0. De plus elle contient une instruction, `printf("hello world");` qui sert à afficher "hello world". La fonction `printf()` qui est utilisée est définie dans la bibliothèque `stdio.h`, c'est pourquoi il est important d'inclure cette bibliothèque. Il n'est ici pas vraiment nécessaire d'inclure `stdlib.h` mais je l'ai fait par habitude (en général, tu en auras besoin).

Notre application affiche donc "hello world" puis retourne 0.

1.2 Les fonctions

Super, on sait maintenant faire un `main` (enfin, à priori...). On peut donc écrire une fonction, et la faire s'exécuter. Oui, mon cher, mais lorsqu'on a un programme, un vrai de vrai, entre les mains, il ne propose en général pas une seule fonction. Tiens, par exemple, un programme que tu connais certainement : Word. Il propose d'écrire du texte (une fonction), d'en supprimer (deux fonctions), d'en copier-coller (trois fonctions), de changer la couleur (quatre fonctions), le mettre en gras (5 fonctions).... Hé oui, on ne va pas coder tout cela comme un porc dans une seule fonction ! Mais alors, comment faire ? Rien de plus simple, mais avant d'aller plus loin, une règle importante je me dois de t'apprendre, mon jeune padawan :



Tout élément appelé dans un code C doit être déclaré au préalable.

Ok, elle est jolie ta phrase, mais en clair, ça veut dire quoi ? Ben ça veut dire que si tu veux utiliser une variable, il faut la déclarer avant, en indiquant son type (est-ce un entier, un caractère, etc..). Nous verrons les types de données juste après (si, si, je te jure). Et pour les fonctions, ça fonctionne... Pareil. Donc pour appeler une fonction, il faut la déclarer au préalable. Ça paraît logique : dans la vie réelle, c'est pareil. Il faut te dire qui est Bob avant que tu ne saches qui il est.

Déclaration de fonction



```
type_de_retour  fonction  ( type_arg_1  nom_arg_1, ... );
```

Un exemple serait le bienvenu. Supposons qu'on veuille faire une fonction somme, qui ajoute deux entiers. Voici la déclaration de la fonction :

```
1 int somme(int a, int b);
```

Ma fonction renvoie un int (un entier), s'appelle somme, et prend en arguments 2 entiers : a et b. On remarque qu'on n'a pas encore codé la fonction : il s'agit de la déclaration. Il est possible de coder la fonction plus loin, une fois celle-ci déclarée, mais pour apprendre nous la coderons directement en la déclarant (on verra ultérieurement comment les pros font ça plus proprement). Bon, on a vu précédemment (donc dans le chapitre 1.1, puisqu'on est dans le 1.2) que LA fonction exécutée quand on lance un programme C, c'est la fonction main. Pour appeler d'autres fonctions, il faut donc faire les appels dans le main (faciiiile !).

Addition de deux nombres :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int somme(int a, int b){
5     int c;          // On declare la variable c
6     c = a+b;        // c vaut la somme a+b
7     return c;       // On renvoie la valeur de c
8 }
9
10 int main(void){
11     int nombre1,nombre2,resultat;
12     nombre1 = 3;
13     nombre2 = 39;
14
15     resultat = somme(nombre1,nombre2);
16
17     printf("%d",resultat); // On affiche la valeur de resultat
18                          // On verra cette notation plus tard
19
20     return 0;
21 }
```

1.3 Compilation

Introduction à gcc

Ouais yahou super j'ai réussi à écrire un code C... Mais maintenant, comment est ce que je le fais fonctionner ? Quand on écrit du code C, on l'écrit dans une langue qu'on comprend (plus ou moins certes). Cependant, l'ordinateur, lui ne sait pas comment interpréter ce langage. Lui, il lui faut son langage, et rien d'autre. Le compilateur va traduire votre code en un langage compréhensible par la machine.

Syntaxe de gcc



```
gcc -Wall -o monprogramme moncode.c
```

Découpons cela pour mieux comprendre ce que je lui demande. **gcc** est le nom du compilateur. **-Wall** est une option demandant au compilateur de t'afficher tout ce qu'il pense être source potentielle d'erreur : elle signifie Warning All. **-o monprogramme** est une option déterminant le nom du programme de sortie (o pour Output). **moncode.c** est le nom du fichier contenant mon code c.

Lancer le programme compilé

Pour lancer le programme, il te suffit de taper cette commande :

```
./monprogramme
```

1.4 Les types de données

Faisons bien, faisons simple. Tu auras besoin de très peu de types de données en C :

int Pour écrire des entiers de taille raisonnable (en général ça suffit)

float Pour écrire des flottants (nombres à virgule)

char Pour écrire des caractères (une seule lettre)

char* Pour écrire une chaîne de caractères (une phrase)

En code :

```
1 int main(void){
2     int entier;
3     char lettre;
4     char* phrase;
5
6     entier    = 42;
7     lettre    = 'a';
8     phrase    = "Coucou ceci est une phrase";
9     return 0;
10 }
```

1.5 Afficher les résultats

Initiation à printf

Il existe une fonction, `printf`, qui permet d'afficher des informations dans le terminal où on se trouve. Elle prend en premier argument la chaîne de caractères à afficher, puis les variables nécessaires. Pour mieux comprendre, regarde la suite :

```
1 int main(void){
2     int a = 22;
3     int b = 20;
4     printf("coucou \n");
5     printf("a vaut %d \n", a);
6     printf("b vaut %d \n", b);
7     printf("a+b vaut %d \n", a+b);
8 }
```

Ceci affiche :

```
osi > ./monprogramme
coucou
a vaut 22
b vaut 20
a+b vaut 42
```

Notations

- `\n` sert à passer à la ligne
- `%d` sert à afficher une variable de type **int**
- `%f` sert à afficher une variable de type **float**
- `%c` sert à afficher une variable de type **char**
- `%s` sert à afficher une variable de type **char***

Syntaxe de printf



```
printf("texte",variables);
```



Lors du TOP n'oublie pas le `\n` : essaye toujours en le mettant et en ne le mettant pas, suivant les exercices, il faut le mettre... ou pas.

1.6 Interactions

Découverte de scanf

Bon, on sait comment afficher des données, il nous reste à permettre à l'utilisateur d'entrer des données. Pour cela, il existe la fonction `scanf`. Cette fonction passe la valeur tapée au clavier par l'utilisateur dans une variable déclarée au préalable.

Syntaxe de scanf



```
scanf("type de donnée", adresse de la variable);
```

Utilisation de scanf

Bon, un exemple sera probablement plus probant. `scanf` fonctionne à l'inverse de `printf`, c'est à dire qu'au lieu de lire les données, il écrit les données dans les variables. Mais sinon, c'est pareil à une exception près. Observe bien :

```
1 int main(void) {  
2     int nombre;  
3     scanf("%d", &nombre);  
4     return 0;  
5 }
```

Non, le `&` n'est pas une faute de frappe : en effet, lorsqu'on tape `"&variable"`, cela correspond à l'adresse mémoire de la variable. T'inquiète pas, on reparlera de la mémoire plus tard, mais n'oublie pas ce `&` très important (sinon tu auras des résultats super aberrants). Tu peux toujours tester la valeur des variables avec un `printf`, ensuite, n'oublie pas.



Attention, pour des chaînes de caractères, c'est plus compliqué. Mais à priori pour l'instant tu n'en as pas besoin.

Un site sympa pour plus d'infos sur **scanf** : <http://xrenault.developpez.com/tutoriels/c/scanf/>

1.7 Travail manuel !



Tu peux à tout moment consulter le manuel d'utilisation d'une fonction, en tapant dans un terminal **man fonction**. Par exemple:
man printf

Chapter 2

Tu peux pas test !

2.1 Découverte des tests

2.1.1 Les tests : qu'est-ce ?

On pourrait en parler longtemps, mais disons tout bêtement qu'un test, c'est une question à laquelle on peut répondre soit par oui, soit par non. En langage informatique, on dira que les deux réponses possibles sont TRUE et FALSE. Le type de données retournées par un test est un booléen. En C, on ne se prend pas la tête, on n'a pas de booléen : on utilise directement des int pour refléter cela. FALSE sera représenté par 0 et TRUE sera représenté par toute valeur différente de 0 (en général, on prend 1).

Définition d'un test

Fonction à un ou plusieurs arguments qui renvoie 0 ou 1.

2.1.2 if, else

Syntaxe de if et else

```
1 if( test 1 ) {  
2     cas 1;  
3 }else if( test 2 ){  
4     cas 2;  
5 }else {  
6     cas 3;  
7 }
```

2.1.3 Les opérateurs

Avant d’aller plus loin, il est important de comprendre quels sont les différents types d’opérateurs, et comment ils fonctionnent.

Arithmétiques

- + Sert à additionner deux nombres
- Sert à soustraire deux nombres
- * Sert à multiplier deux nombres
- / Sert à diviser deux nombres (*attention à la division par 0*)

Comparaison

- == Signifie "égal" (*attention à bien mettre deux "="*)
- != Signifie "différent"
- > Signifie "strictement plus grand que"
- < Signifie "strictement plus petit que"
- >= Signifie "supérieur ou égal à"
- <= Signifie "inférieur ou égal à"

Logiques

- || Signifie "ou"
- && Signifie "et"
- ! Permet d’obtenir la négation d’un résultat

Exemples

a==1 && b != 3 a vaut 1 et b est différent de 3

a>1 || ! (b==3 || b>5) a est strictement supérieur à 1 ou (b vaut 3 ou b>5) est faux

(a==1 && b!=3) || !(a>2 && b<=4) (a vaut 1 et b est différent de 3) ou (a est plus grand que 2 et b plus petit ou égal à 4) est faux

2.1.4 Switch

Bon, supposons maintenant qu'on veuille tester plein de valeurs possibles pour une variable. On va utiliser **switch case**. La structure est simple :

Structure de switch

```
1 switch(variable1){  
2     case Valeur1: (cas 1)  
3         break;  
4     case Valeur2: (cas 2)  
5         break;  
6     ...  
7     default : (cas par default)  
8 }
```

Exemple d'utilisation

Switch est souvent utilisé pour faire des menus (suivant la valeur appuyée par l'utilisateur, on va faire telle ou telle chose). Voici un exemple simple :

```
1 switch(a){  
2     case 1: printf("bonjour monsieur\n");  
3         break;  
4     case 2: printf("vous avez tape 2\n");  
5         break;  
6     default: printf("je n'ai pas compris\n");  
7 }
```



2.2 Cherchons la boucle... d'or ?

2.2.1 while

On a vu précédemment que les tests peuvent servir à émettre des conditions. Ils sont également utiles lorsqu'on veut faire des boucles. En effet, on peut s'en servir dans une boucle **while** (tant que). Par exemple, on peut dire que "tant que n est inférieur strict à 3, on fait quelque chose".

Syntaxe de while



```
while( test ){ commandes }
```

Exemple

Utilisons **while** pour afficher la table de 9 :

```
1 int main(void){
2     int n=0;
3     while( n < 10 ){
4         printf("%d \n", 9*n);
5         n=n+1;
6     }
7     return 0;
8 }
```

Cet exemple affichera les multiples de 9 de 0 à 81. Une fois que n=10, il s'arrête.



Attention à ne pas mettre une condition dans le while qui se vérifie indéfiniment ! Le programme compilera quand même, mais on obtient alors une boucle infinie. Par exemple :

```
1 int n = 2;
2 while(n!=3){
3     n = n + 2;
4 }
```

n va valoir 2, 4, 6, ... et ne passera jamais par 3 : n est toujours différent de 3, donc on continue indéfiniment.

2.2.2 for

Il existe un autre type de boucle, les boucles "for" (pour). Ces boucles sont à utiliser lorsqu'on veut exécuter une action un nombre n de fois. Dans ce cas, on dira qu'on veut l'exécuter pour i allant de 1 à n (ou de 0 à $n-1$). Utilisons cela pour afficher à nouveau la table de 9 :

```
1 int main(void){
2     int i;
3     for(i=0; i<10; i++){
4         printf("%d \n", 9*i);
5     }
6     return 0;
7 }
```

Cet exemple a les mêmes effets que l'exemple pour while. Dans une boucle for, on indique :

- La condition de départ : $i=0$
- La condition qui, lorsqu'elle n'est plus vérifiée, arrête tout : $i<10$
- Ce qu'on fait entre chaque étape : $i++$ (on augmente la valeur de i de 1)
- Ce qu'on fait dans la boucle : le printf

Attention : on sépare ces indications par des point-virgules, et non par de simples virgules (sinon ça marche pas) ! De même, n'oublie pas de déclarer la variable qui intervient dans le for (ici, on indique bien "int i" au début).



Attention à ne pas mettre une condition dans le for qui se vérifie indéfiniment ! Le programme compilera quand même, mais on obtient alors une boucle infinie. Par exemple :

```
1 int n;
2 for(n=2; n>1; n++){
3     printf("Zut, boucle infinie...");
4 }
```

n va valoir 2, 3, 4, ... et ne sera jamais inférieur ou égal à 1, donc on continue indéfiniment.

Comment ? Tu as déjà vu ça dans la partie précédente ? Alors c'est que ça doit vraiment être important, si je le mets deux fois... Peut être que ça arrive bien plus souvent que tu ne le penses. Te voilà, cher lecteur, doublement averti.

Chapter 3

Tableaux à la Picasso

Explications

Un tableau sert à ranger des trucs du même type dans des cases. Bref, dans la vie réelle je pourrais modéliser mon tableau par une étagère, et chaque étage correspond à une case.

Syntaxe de déclaration d'un tableau



<code>type_de_variable nom_du_tableau [nombre_de_cases]</code>
--

Exemple

Soit A une armoire de taille 4 étages. Sur chaque étage, on a un nombre n d'objets dépendant de l'étage. Utilisons un tableau pour indiquer le nombre d'objets par étage. Voici la répartition des objets.

- Etage 0 : 3 objets
- Etage 1 : 2 objets
- Etage 2 : 5 objets
- Etage 3 : 0 objets

```
1 int main(void){
2     int i; // pour ma boucle finale
3     int etagere[4]; // Tableau d'entiers de 4 cases : 0,1,2,3
4
5     etagere[0] = 3;
6     etagere[1] = 2;
7     etagere[2] = 5;
8     etagere[3] = 0;
9
10    for(i=0; i<4; i++){
11        printf("Il y a %d elements sur l'etagere %d", etagere[i] , i );
12    }
13    return 0;
14 }
```

Chapter 4

Pointe pas du doigt !

4.1 Introduction aux pointeurs

4.1.1 Que c'est moche !

Tu en as peut être déjà entendu parler, et je suppose que, en commençant ce chapitre, tu te dis "vouha les pointeurs c'est trop compliqué, en plus ça sert à rien, c'est horrible !". Bon, commençons par t'informer d'une chose importante : les informaticiens ne sont pas tous des masochistes. Donc si il y a une notion, en général elle n'est pas inutile.

4.1.2 Mais comment ça marche alors ?

Il faut bien comprendre comment fonctionne la mémoire de l'ordinateur pour comprendre les pointeurs. La mémoire est organisée en cases. Chaque case est située à une adresse donnée. On peut se représenter cela avec une série de boîtes aux lettres posées côte à côte. Le pointeur va indiquer l'adresse de la case correspondant à ce que l'on désire. Pour déclarer un pointeur voici ce que l'on écrit :

```
1 int* doigt;
```

En écrivant ceci, on déclare un pointeur sur un int, nommé doigt. Il faudra cependant allouer de la mémoire pour cet int sur lequel on pointe, ce que j'explique dans la partie suivante (sinon, le programme plante et affiche: *segmentation fault*). Mais revenons au pointeur. Pour définir sa valeur, on écrit :

```
1 *doigt = 3;
```

Ici, doigt est une adresse pointant vers une case de valeur 3. On a donc vu que "*" permet de définir un pointeur. Mais comment récupérer l'adresse où est stockée une variable ? Il suffit d'utiliser "&". On a donc :

```
1 int main(void){  
2     int n=5; // on declare une variable  
3     int* adresse_n; // on declare un pointeur vers un int  
4     adresse_n = &n; // on fait pointer le pointeur vers notre variable  
5     *adresse_n = 10; // on modifie la valeur pointee par adresse_n  
6     printf("%d \n",n); // on affiche la valeur de n  
7     return 0;  
8 }
```

Et ce programme renvoie... 10. En effet, la valeur de n a été modifiée, car on a modifié la valeur sur laquelle pointait adresse_n.

4.2 Pour un malloc libre !

4.2.1 malloc

Explication

Parfois, on désire utiliser davantage qu'un simple entier. Lors de l'utilisation de tableau, par exemple, ou de listes. Dans ce cas, il faut allouer la place en mémoire pour pouvoir placer nos nombres. Par exemple, si je veux écrire une liste de 5 entiers, il me faudra 5 fois la place d'un entier en mémoire. Dans le cas d'un tableau, ces entiers sont côte à côte en mémoire. Il suffit donc de créer un pointeur qui pointera vers la première case du tableau, et de réserver libres les 4 cases suivantes, plus la première. Afin d'obtenir un tel espace, on utilise **malloc**, qui vient de Memory Allocate (allouer de la mémoire).

Syntaxe



```
pointeur = malloc ( nombre de donnees * sizeof( type de donnees ));
```

Exemple

On va allouer la place à 5 entiers dans la mémoire :

```
1 int* pointeur;  
2 pointeur = malloc( 5 * sizeof( int ) );
```

4.2.2 free

Explication

Bon, on sait allouer de la place en mémoire. Cependant, il se trouve que tout le monde n'en a pas une grosse (tu en sais quelque chose), de mémoire. Et donc, à force d'utiliser de la place en mémoire pour stocker, on finit par ne plus avoir de place. Alors que certaines informations, encore en mémoire, ne servent plus à rien.

C'est là qu'intervient **free**. Cette fonction permet de libérer la mémoire occupée par l'élément pointé.

Syntaxe



```
free(pointeur);
```

Exemple

Reprenons l'élément précédent et libérons le :

```
1 int* pointeur;  
2 pointeur = malloc( 5 * sizeof( int ) );  
3 free(pointeur);
```



Une fois le free effectué, il est impossible d'avoir à nouveau accès aux données effacées !



Bon, c'est très mal de te dire cela, mais si ça peut t'aider... Pour le TOP, comme le but est juste de faire tourner rapidement une application toute simple, tu ne risques pas de saturer la mémoire si tu ne fais pas de **free** : ce n'est donc pas grave si tu n'en mets pas lors de ton TOP. Par contre, ce sera bien plus grave si tu les oublies dans tes projets.

4.2.3 Démonstration de l'utilité du free

Prenons un code qui fait une boucle infinie et qui, à chaque itération, va allouer beaucoup de place en mémoire. Ceci aura, au bout de quelques secondes, le même effet qu'un autre programme normal sans boucle infini avec des **malloc** et pas de **free**. Voici mon programme :

```
1 int main(void) {  
2     int* pointeur;  
3     while(1) {  
4         pointeur = malloc(10000*sizeof(int));  
5     }  
6     return 0;  
7 }
```

Voici le résultat sur mon ordi en 10 secondes :

Avant :

PhysMem: 961M used, 1087M free.

Après :

PhysMem: 1159M used, 889M free.

100 M de ram utilisés par le programme en 10 secondes (et bien évidemment, ça ne cesse d'augmenter). Bon, truc sympa à savoir : lorsqu'un programme s'arrête, tout ce qui était contenu dans les variables internes au programme est libéré de la mémoire. Maintenant, ajoutons le **free** au programme :

```

1 int main(void){
2     int* pointeur;
3     while(1){
4         pointeur = malloc(10000*sizeof(int));
5         free(pointeur);
6     }
7     return 0;
8 }

```

Voici le résultat sur mon ordi en 10 secondes :

Avant :

PhysMem: 961M used, 1087M free.

Après :

PhysMem: 960M used, 1088M free.

Oh ! Il a même fait de la place ! Non non, rien à voir, c'est juste qu'un ordi, en général, ça fait pas tourner un seul programme. Enfin voilà, on voit bien que le free est fort utile.

Donc n'oublie pas : un **malloc** implique un **free**.

4.3 Exemple simple sur les pointeurs

Voici un petit exemple qui stocke les multiples de 9 en mémoire, puis les affiche.

```

1 int main(void){
2     int i;
3     int* multiples;
4     int* pointeur2;
5     multiples = malloc(42*sizeof(int)); // On alloue l'espace
6     pointeur2 = multiples;
7
8     /* Methode 1 de parcours : on stocke */
9     for(i=0;i<42;i++){
10         *pointeur2=9*i;
11         pointeur2++;
12     }
13
14     /* Methode 2 de parcours : on affiche */
15
16     for(i=0;i<3;i++){
17         printf("%d\n",multiples[i]);
18     }
19     free(multiples);
20     return 0;
21 }

```

Mais que se passe-t-il ici ? Dans la méthode 1, on modifie la valeur pointée par pointeur2, puis on déplace ce pointeur pour pointer sur l'adresse suivante, et on recommence. Dans la méthode 2, on affiche multiples[0] puis multiples[1], qui est en fait ce qui se trouve "après" multiples[0] dans la mémoire (souviens toi de l'explication de la mémoire organisée en cases). En d'autres termes,

`multiples[1] = *(multiples + 1);`

4.4 Les listes

typedef struct

Rah la la ces fichues listes.. Bon, explication. On a vu que les pointeurs, ça permet de faire des tableaux. Or, les tableaux contiennent des données du même type. Supposons que j'ai envie de me faire une liste de mes potes, en y rangeant leur nom, prénom, sexe, numéro de téléphone. Allez, c'est parti, découvrons le **typedef struct** !

Hein ? Le kézako ? Facile : struct permet de définir une structure, et typedef définit cette structure comme un type de données à part entière. On va créer un type copain :

```
1 typedef struct copain{
2     char* nom;
3     char* prenom;
4     char sexe;
5     int tel;
6 }copain;
```

Pour interagir avec un de mes copains (petit canaillou), regarde l'explication par l'exemple :

```
1 copain jean;
2 jean.nom = "Dupont";
3 jean.prenom = "Jean";
4 jean.sexe = 'm';
5 jean.tel = 0142424242;
```

On peut également interagir en utilisant un pointeur vers la structure copain :

```
1 copain* john;
2 john=malloc(sizeof(copain));
3 john->nom = "Smith";
4 john->prenom = "John";
5 john->sexe = 'm';
6 john->tel = 0242424242;
```

La liste

Maintenant, comme on veut faire une liste de copains, on va dire à chacun d'entre eux de pointer du doigt le suivant : en partant du premier d'entre eux, et en suivant le chemin qu'ils indiquent, on passera par tous ! C'est ce qu'on appelle une liste chaînée.

```
1 typedef struct copain_pointe{
2     copain cop;
3     copain* suivant;
4 } copain_pointe;
```

copain_pointe est un type qui contient un copain ainsi que l'adresse du copain suivant. C'est la base d'une liste chaînée de copains.



Tout élément d'une liste chaînée contient des informations sur lui-même et un pointeur vers l'élément suivant de la liste.

Il est important pour toi de comprendre comment fonctionnent les différentes opérations sur les listes. Je ne noterai pas ici les fonctions permettant les opérations sur les listes, car elles sont à découvrir en TD/TP, ce qui te sera bien plus profitable. Libre à toi d'écrire les fonctions de base sur cette page, cependant.

Chapter 5

Une main mieux employée

5.1 Pourquoi ? Mais pourquoiiiii ?

”Hey, c’est bon, on commence à le savoir que c’est `int main(void)` qu’il faut taper hein...”
Oui, bon, d’accord, je te l’ai dit au moins 20 fois dans ce poly, il faut écrire `int main(void)`.
Le problème (QUOI ? y a un problème ? ?), c’est que pour l’instant, on n’a jamais pu donner d’arguments directement en lançant le programme. Et ça, ce serait classe. Exemple de ligne sympatoche :

```
osi> ./somme 3 5
8
```

Ooooooh trop classe ! Et comment on fait ça ? Ben on tape plus `”int main (void)”` mais autre chose... Hé me frappe pas, tu vas vite comprendre pourquoi je t’ai pas donné la vraie structure en premier lieu.

5.2 Retour sur la structure du main

Le main peut prendre des argument. Vlan. Ca, c’est dit. Voici la véritable structure du main, pour tes yeux ébahis :



```
int main( int argc, char** argv){ [...]; return 0; }
```

Vouahou c’est moche ! Bon, t’en fais pas, c’est tout bête. Lorsqu’on lance le programme, le main va avoir deux arguments : `argc`, un `int`, et `argv`, un `char**`. Bon, `argc`, c’est un nombre, qui correspond au nombre d’arguments données au programme + 1 : le nom du programme est compté comme un argument. En clair, il correspond au nombre de mots placés sur la ligne.

Un `char**`, mais qu’est-ce que c’est que cette bête là ? ? ? Décomposons :

un `char` est un caractère

un `char*` est un tableau de caractère, donc un mot ou une phrase

un `char**` sera donc un tableau de tableaux de caractères, donc un tableau de mots ou phrases : ici, ce seront les différents mots placés sur la ligne, stockés dans un tableau.

Démonstration par l'exemple

```
1 int main(int argc, char** argv){
2     printf("argc vaut : %d \n", argc);
3     printf("argv[0] vaut : %s \n", argv[0]);
4     printf("argv[1] vaut : %s \n", argv[1]);
5     printf("argv[2] vaut : %s \n", argv[2]);
6
7     return 0;
8 }
```

```
osi> ./affiche haha 25
argc vaut : 3
argv[0] vaut : ./affiche
argv[1] vaut : haha
argv[2] vaut : 25
```

5.3 Utilisation du main

Voilà, on sait maintenant récupérer des informations en ligne de commande. Cependant, elles sont entrées au format de `char*` : de chaîne de caractères. Pour une addition de deux nombres, si on obtient les nombres en format de chaînes de caractères c'est pas super classe. Il faut donc les convertir en nombres. La fonction qui permet ceci est : **atoi**. Cette fonction convertit de l'ASCII vers un int. Ascii TO Int. Easy. Il existe une foule de fonctions du même genre, comme `atof`, `atol` (les opticiens), etc... Il te suffit de faire un petit **man atoi** pour savoir ca.



Voici la structure de base que tu DOIS savoir pour passer le TOP. Ici, on va exécuter une addition.

```
1 int main(int argc, char** argv){
2     if(argc!=3) { return 1; } //on renvoie 1 si on n'a pas 2 nombres
3     a additionner
4     int resultat;
5     resultat = atoi(argv[1]) + atoi(argv[2]);
6
7     printf("%d\n", resultat); // ne pas oublier le \n
8     return 0;
9 }
```

Bonus : la pyramide d'étoiles



Ceci est un exercice qui tombe fréquemment au TOP.

Enoncé : Faire un programme qui affiche une pyramide d'étoiles de taille n , passée en argument.

Corrigé :

```
1 int main(int argc, char** argv){
2     int i,j,n;
3     if(argc!=2){return 1;} // si on n'a pas la taille en argument, on
        renvoie une erreur
4     n=atoi(argv[1]);
5
6     for(i=0;i<n;i++){
7         for(j=0;j<i+1;j++){
8             printf("*");
9         }
10        printf("\n");
11    }
12 }
```

Et, pour te montrer qu'il y a plein de façons d'écrire une même fonction en C :

```
1 int main(int argc, char** argv){
2     int i,n;
3     if(argc!=2) { return 1;}
4     n=atoi(argv[1]);
5     char* etoiles;
6     etoiles=calloc(sizeof(char),n+1);
7     for(i=0;i<n;i++){
8         etoiles[i]='*';
9         puts(etoiles);
10    }
11    return 0;
12 }
```

Note : la seconde s'exécute plus de 10 fois plus vite que la première sur mon ordi. D'où les idées d'optimisation de code...

Fin

Merci à tous les lecteurs, relecteurs, prélecteurs, postlecteurs, etc... qui ont pu m'aider à réaliser ce poly, ainsi bien sûr qu'à mes fillots qui m'ont donné l'idée et la motivation pour le faire. N'hésitez pas à me contacter si vous avez des questions sur des points obscurs de ce poly ou du C en général.

Bon courage pour tes études !

